CSE 241 Lecture 7

Adopted from the lecture slides of the book:
Absolute C++ by Walter Savitch, Kenrick Mock

Learning Objectives

- Constructors
  - Definitions
  - Calling
- More Tools
  - `const` parameter modifier
  - Inline functions
  - Static member data
- Vectors
  - Introduction to vector class

Constructors

- Initialization of objects
  - Initialize some or all member variables
  - Other actions possible as well
- A special kind of member function
  - Automatically called when object declared
- Very useful tool
  - Key principle of OOP

Constructor Definitions

- Constructors defined like any member function
  - Except:
    1. Must have same name as class
    2. Cannot return a value; not even void!

Constructor Definition Example

- Class definition with constructor:
  -
```
class DayOfYear
{
public:
     DayOfYear(int monthValue, int dayValue);
       //Constructor initializes month and day
     void input();
     void output();
     …
private:
     int month;
     int day;
}
```

Constructor Notes

- Notice name of constructor: `DayOfYear`
  - Same name as class itself!
- Constructor declaration has no return-type
  - Not even `void`!
- Constructor in public section
  - It's called when objects are declared
  - If private, could never declare objects!

Calling Constructors

- Declare objects:

```
DayOfYear date1(7, 4), date2(5, 5);
```

- Objects are created here
  - Constructor is called
  - Values in parens passed as arguments to constructor
  - Member variables `month`, `day` initialized:
    `date1.month` → `7` `date2.month` → `5`
    `date1.dat` → `4` `date2.day` → `5`

Constructor Equivalency

- Consider:
  - `DayOfYear date1, date2`

    ```
    date1.DayOfYear(7, 4);    // ILLEGAL!
    date2.DayOfYear(5, 5);    // ILLEGAL!
    ```

- Seemingly OK…
  - CANNOT call constructors like other member functions!

Constructor Code

- Constructor definition is like all other member functions:

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
{
    month = monthValue;
    day = dayValue;
}
```

- Note same name around ::
  - Clearly identifies a constructor
- Note no return type
  - Just as in class definition

Alternative Definition

- Previous definition equivalent to:

```
DayOfYear::DayOfYear(int monthValue,int dayValue)
          : month(monthValue), day(dayValue)  ←
{…}
```

- Third line called "Initialization Section"
- Body left empty
- Preferable definition version

Constructor Additional Purpose

- Not just initialize data
- Body doesn't have to be empty
  - In initializer version
- Validate the data!
  - Ensure only appropriate data is assigned to class private member variables
  - Powerful OOP principle

Overloaded Constructors

- Can overload constructors just like other functions
- Recall: a signature consists of:
  - Name of function
  - Parameter list
- Provide constructors for all possible argument-lists
  - Particularly "how many"

Class with Constructors Example:
Display 7.1 Class with Constructors (1 of 3)

Display 7.1 Class with Constructors

```
#include <iostream>
#include <cstdlib> //for exit
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    //Initializes the month and day to arguments.

    DayOfYear(int monthValue);
    //Initializes the date to the first of the given month.

    DayOfYear( );
    //Initializes the date to January 1.

    void input( );
    void output( );
    int getMonthNumber( );
    //Returns 1 for January, 2 for February, etc.
```

This definition of `DayOfYear` *is an improved version of the class* `DayOfYear` *given in Display 6.4.*

default constructor

Class with Constructors Example:
Display 7.1 Class with Constructors (2 of 3)

```
    int getDay( );
private:
    int month;
    int day;
    void testDate( );
};

int main( )
{
    DayOfYear date1(2, 21), date2(5), date3;
    cout << "Initialized dates:\n";
    date1.output( ); cout << endl;
    date2.output( ); cout << endl;
    date3.output( ); cout << endl;

    date1 = DayOfYear(10, 31);
    cout << "date1 reset to the following:\n";
    date1.output( ); cout << endl;
    return 0;
}

DayOfYear::DayOfYear(int monthValue, int dayValue)
                          : month(monthValue), day(dayValue)
{
    testDate( );
}
```

This causes a call to the default constructor. Notice that there are no parentheses.

an explicit call to the constructor `DayOfYear::DayOfYear`

Class with Constructors Example:
Display 7.1 Class with Constructors (3 of 3)

Display 7.1 Class with Constructors

```
DayOfYear::DayOfYear(int monthValue) : month(monthValue), day(1)
{
    testDate( );
}

DayOfYear::DayOfYear( ) : month(1), day(1)
{/*Body intentionally empty.*/}

//uses iostream and cstdlib:
void DayOfYear::testDate( )
{
    if ((month < 1) || (month > 12))
    {
        cout << "Illegal month value!\n";
        exit(1);
    }
    if ((day < 1) || (day > 31))
    {
        cout << "Illegal day value!\n";
        exit(1);
    }
}
```

<Definitions of the other member functions are the same as in Display 6.4.>

Sample Dialogue

```
Initialized dates:
February 21
May 1
January 1
date1 reset to the following:
October 31
```

Constructor with No Arguments

- Can be confusing
- Standard functions with no arguments:
  - Called with syntax: `callMyFunction();`
    - Including empty parentheses
- Object declarations with no "initializers":
  - `DayOfYear date1;  // This way!`
  - `DayOfYear date(); // NO!`
    - What is this really?
    - Compiler sees a function declaration/prototype!
    - Yes! Look closely!

Explicit Constructor Calls

- Can also call constructor AGAIN
  - After object declared
    - Recall: constructor was automatically called then
  - Can call via object's name; standard member function call
- Convenient method of setting member variables
- Method quite different from standard member function call

Explicit Constructor Call Example

- Such a call returns "anonymous object"
  - Which can then be assigned
  - **In Action**:
    `DayOfYear holiday(7, 4);`
    - Constructor called at object's declaration
    - Now to "re-initialize":
      `holiday = DayOfYear(5, 5);`
      - Explicit constructor call
      - Returns new "anonymous object"
      - Assigned back to current object

Default Constructor

- Defined as: constructor w/ no arguments
- One should always be defined
- Auto-Generated?
  - Yes & No
  - If no constructors AT ALL are defined → Yes
  - If any constructors are defined → No
- If no default constructor:
  - Cannot declare: `MyClass myObject;`
    - With no initializers

Class Type Member Variables

- Class member variables can be any type
  - Including objects of other classes!
  - Type of class relationship
    - Powerful OOP principle
- Need special notation for constructors
  - So they can call "back" to member object's constructor

Class Member Variable Example:
Display 7.3 A Class Member Variable (1 of 5)

Display 7.3 A Class Member Variable

```
#include <iostream>
#include<cstdlib>
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    DayOfYear(int monthValue);
    DayOfYear( );
    void input( );
    void output( );
    int getMonthNumber( );
    int getDay( );
private:
    int month;
    int day;
    void testDate( );
};
```

The class **DayOfYear** *is the same as in Display 7.1, but we have repeated all the details you need for this discussion.*

Class Member Variable Example:
Display 7.3 A Class Member Variable (2 of 5)

```
class Holiday
{
public:
    Holiday( );//Initializes to January 1 with no parking enforcement
    Holiday(int month, int day, bool theEnforcement);
    void output( );
private:
    DayOfYear date;
    bool parkingEnforcement;//true if enforced
};

int main( )
{
    Holiday h(2, 14, true);
    cout << "Testing the class Holiday.\n";
    h.output( );

    return 0;
}

Holiday::Holiday( ) : date(1, 1), parkingEnforcement(false)
{/*Intentionally empty*/}

Holiday::Holiday(int month, int day, bool theEnforcement)
                    : date(month, day), parkingEnforcement(theEnforcement)
{/*Intentionally empty*/}
```

member variable of a class type

Invocations of constructors from the class `DayOfYear`.

(continued)

Class Member Variable Example:
Display 7.3 A Class Member Variable (3 of 5)

Display 7.3 A Class Member Variable

```
void Holiday::output( )
{
    date.output( );
    cout << endl;
    if (parkingEnforcement)
        cout << "Parking laws will be enforced.\n";
    else
        cout << "Parking laws will not be enforced.\n";
}

DayOfYear::DayOfYear(int monthValue, int dayValue)
                            : month(monthValue), day(dayValue)
{
    testDate( );
}
```

Class Member Variable Example:
Display 7.3 A Class Member Variable (4 of 5)

```
//uses iostream and cstdlib:
void DayOfYear::testDate( )
{
    if ((month < 1) || (month > 12))
    {
        cout << "Illegal month value!\n";
        exit(1);
    }
    if ((day < 1) || (day > 31))
    {
        cout << "Illegal day value!\n";
        exit(1);
    }
}

//Uses iostream:
void DayOfYear::output( )
{
    switch (month)
    {
        case 1:
            cout << "January "; break;
        case 2:
            cout << "February "; break;
        case 3:
            cout << "March "; break;
          .
          .
          .
```

The omitted lines are in Display 6.3, but they are obvious enough that you should not have to look there.

Class Member Variable Example:
Display 7.3 A Class Member Variable (5 of 5)

Display 7.3 A Class Member Variable

```
        case 11:
            cout << "November "; break;
        case 12:
            cout << "December "; break;
        default:
            cout << "Error in DayOfYear::output. Contact software vendor.";
    }

    cout << day;
}
```

SAMPLE DIALOGUE

```
Testing the class Holiday.
February 14
Parking laws will be enforced.
```

Parameter Passing Methods

- Efficiency of parameter passing
  - Call-by-value
    - Requires copy be made → Overhead
  - Call-by-reference
    - Placeholder for actual argument
    - Most efficient method
  - Negligible difference for simple types
  - For class types → clear advantage
- Call-by-reference desirable
  - Especially for "large" data, like class types

The `const` Parameter Modifier

- Large data types (typically classes)
  - Desirable to use pass-by-reference
  - Even if function will not make modifications
- Protect argument
  - Use constant parameter
    - Also called constant call-by-reference parameter
  - Place keyword `const` before type
  - Makes parameter "read-only"
  - Attempt to modify parameter results in compiler error

Use of `const`

- All-or-nothing
- If no need for function modifications
  - Protect parameter with const
  - Protect ALL such parameters
- This includes class member function parameters

Inline Functions

- For non-member functions:
  - Use keyword `inline` in function declaration and function heading
- For class member functions:
  - Place implementation (code) for function IN class definition → automatically inline
- Use for very short functions only
- Code actually inserted in place of call
  - Eliminates overhead
  - More efficient, but only when short!

Inline Member Functions

- Member function definitions
  - Typically defined separately, in different file
  - Can be defined IN class definition
    - Makes function "in-line"
- Again: use for very short functions only
- More efficient
  - If too long → actually less efficient!

Member Initializers

- C++11 supports a feature called member initialization
  - This feature allows you to set default values for member variables

```
class Coordinate
{
  public:
        Coordinate();
  private:
        int x=1;
        int y=2;
};
Coordinate::Coordinate()
{ }
```

Member Initializers

```
Coordinate c1;
```

Initializes `c1.x` to `1` and `c1.y` to `2`

Constructor Delegation

- C++11 allows one constructor to invoke another

```
Coordinate::Coordinate(int xval, int yval) : x(xval), y(yval)
{ }

Coordinate::Coordinate() : Coordinate(99,99)
{ }
```

- The default constructor invokes the constructor to initialize `x` and `y` to `99,99`

Static Members

- Static member variables
  - All objects of class "share" one copy
  - One object changes it → all see change
- Useful for "tracking"
  - How often a member function is called
  - How many objects exist at given time
- Place keyword `static` before type

Static Functions

- Member functions can be static
  - If no access to object data needed
  - And still "must" be member of the class
  - Make it a static function
- Can then be called outside class
  - From non-class objects:
    - E.g., `Server::getTurn();`
  - As well as via class objects
    - Standard method: `myObject.getTurn();`
- Can only use static data, functions!

Static Members Example:
Display 7.6 Static Members (1 of 4)

Display 7.6 Static Members

```
#include <iostream>
using namespace std;

class Server
{
public:
    Server(char letterName);
    static int getTurn( );
    void serveOne( );
    static bool stillOpen( );
private:
    static int turn;
    static int lastServed;
    static bool nowOpen;
    char name;
};

int Server:: turn = 0;
int Server:: lastServed = 0;
bool Server::nowOpen = true;
```

Static Members Example:
Display 7.6 Static Members (2 of 4)

```
int main( )
{
    Server s1('A'), s2('B');
    int number, count;
    do
    {
        cout << "How many in your group? ";
        cin >> number;
        cout << "Your turns are: ";
        for (count = 0; count < number; count++)
            cout << Server::getTurn( ) << ' ';
        cout << endl;
        s1.serveOne( );
        s2.serveOne( );
    } while (Server::stillOpen( ));

    cout << "Now closing service.\n";

    return 0;
}
```

Static Members Example:
Display 7.6 Static Members (3 of 4)

Display 7.6 Static Members

```
Server::Server(char letterName) : name(letterName)
{/*Intentionally empty*/}

int Server::getTurn( )
{
    turn++;
    return turn;
}
bool Server::stillOpen( )
{
    return nowOpen;
}

void Server::serveOne( )
{
    if (nowOpen && lastServed < turn)
    {
        lastServed++;
        cout << "Server " << name
            << " now serving " << lastServed << endl;
     }
```

Since **getTurn** *is static, only static members can be referenced in here.*

Static Members Example:
Display 7.6 Static Members (4 of 4)

```
58        if (lastServed >= turn) //Everyone served
59            nowOpen = false;
60    }
```

SAMPLE DIALOGUE

How many in your group? **3**
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? **2**
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? **0**
Your turns are:
Server A now serving 5
Now closing service.

Vectors

- Vector Introduction
  - Recall: arrays are fixed size
  - Vectors: "arrays that grow and shrink"
    - During program execution
  - Formed from Standard Template Library(STL)
    - Using template class

Vector Basics

- Similar to array:
  - Has base type
  - Stores collection of base type values
- Declared differently:
  - Syntax: `vector<Base_Type>`
    - Indicates template class
    - Any type can be "plugged in" to `Base_Type`
    - Produces "new" class for vectors with that type
  - Example declaration:
    `vector<int> v;`

Vector Use

- `vector<int> v;`
  - "v is vector of type int"
  - Calls class default constructor
    - Empty vector object created
- Indexed like arrays for access
- But to add elements:
  - Must call member function `push_back`
- Member function `size()`
  - Returns current number of elements

Vector Example:
Display 7.7 Using a Vector (1 of 2)

Display 7.7 Using a Vector

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size( ) = " << v.size( ) << endl;
        cin >> next;
    }
```

Vector Example:
Display 7.7 Using a Vector (2 of 2)

```
    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

SAMPLE DIALOGUE

Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 –1
2 added. v.size = 1
4 added. v.size = 2
6 added. v.size = 3
8 added. v.size = 4
You entered:
2 4 6 8

Vector Efficiency

- Member function `capacity()`
  - Returns memory currently allocated
  - Not same as `size()`
  - Capacity typically > size
    - Automatically increased as needed
- If efficiency critical:
  - Can set behaviors manually
    - `v.reserve(32);  //sets capacity to 32`
    - `v.reserve(v.size()+10);  //sets capacity to 10 more than size`

Summary 1

- Constructors: automatic initialization of class data
  - Called when objects are declared
  - Constructor has same name as class
- Default constructor has no parameters
  - Should always be defined
- Class member variables
  - Can be objects of other classes
    - Require initialization-section

Summary 2

- Constant call-by-reference parameters
  - More efficient than call-by-value
- Can `inline` very short function definitions
  - Can improve efficiency
- Static member variables
  - Shared by all objects of a class
- Vector classes
  - Like: "arrays that grow and shrink"